NEUZZ: Efficient Fuzzing with Neural Program Smoothing

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray and Suman Jana
Department of Computer Science
Columbia University, New York, USA
{dongdong, kpei, junfeng, rayb and suman}@cs.columbia.edu , dave.epstein@columbia.edu

***Abstract*—Fuzzing has become the de facto standard technique for finding software vulnerabilities. However, even state-of-the-art fuzzers are not very efficient at finding hard-to-trigger software bugs. Most popular fuzzers use evolutionary guidance to generate inputs that can trigger different bugs. Such evolutionary algorithms, while fast and simple to implement, often get stuck in fruitless sequences of random mutations. Gradient-guided optimization presents a promising alternative to evolutionary guidance. Gradient-guided techniques have been shown to significantly outperform evolutionary algorithms at solving high-dimensional structured optimization problems in domains like machine learning by efficiently utilizing gradients or higher-order derivatives of the underlying function.**

However, gradient-guided approaches are not directly applicable to fuzzing as real-world program behaviors contain many discontinuities, plateaus, and ridges where the gradient-based methods often get stuck. We observe that this problem can be addressed by creating a smooth surrogate function approximating the target program's discrete branching behavior. In this paper, we propose a novel program smoothing technique using surrogate neural network models that can incrementally learn smooth approximations of a complex, real-world program's branching behaviors. We further demonstrate that such neural network models can be used together with gradient-guided input generation schemes to significantly increase the efficiency of the fuzzing process.

Our extensive evaluations demonstrate that NEUZZ significantly outperforms 10 state-of-the-art graybox fuzzers on 10 popular real-world programs both at finding new bugs and achieving higher edge coverage. NEUZZ found 31 previously unknown bugs that other fuzzers failed to find in 10 real-world programs and achieved 3× more edge coverage than all of the tested graybox fuzzers for 24 hours running. Furthermore, NEUZZ also outperformed existing fuzzers on both LAVA-M and DARPA CGC bug datasets.

I. INTRODUCTION

Fuzzing has become the de facto standard technique for finding software vulnerabilities [1], [2]. The fuzzing process involves generating random test inputs and executing the target program with these inputs to trigger potential security vulnerabilities [3]. Due to its simplicity and low performance overhead, fuzzing has been very successful at finding different types of security vulnerabilities in many real-world programs [4], [5], [6], [7], [8], [9]. Despite their tremendous promise, popular fuzzers, especially for large programs, often tend to get stuck trying redundant test inputs and struggle to find security vulnerabilities hidden deep within program logic [10], [11], [12].

Conceptually, fuzzing is an optimization problem whose goal is to find program inputs that maximize the number of vulnerabilities found within a given amount of testing time [13]. However, as security vulnerabilities tend to be sparse and erratically distributed across a program, most fuzzers aim to test as much program code as they can by maximizing some form of code coverage (*e.g.*, edge coverage) to increase their chances of finding security vulnerabilities. Most popular fuzzers use evolutionary algorithms to solve the underlying optimization problem, *i.e.*, generating new inputs that maximize code coverage [1], [8], [9], [14]. Evolutionary optimization starts from a set of seed inputs, applies random mutations to the seeds to generate new test inputs, executes the target program for these inputs, and only keeps the promising new inputs (*e.g.*, those that achieve new code coverage) as part of a corpus for further mutation. However, as the input corpus gets larger, the evolutionary process becomes increasingly less efficient at reaching new code locations.

One of the main limitations of evolutionary optimization algorithms is that they cannot leverage the structure (*i.e.*, gradients or other higher-order derivatives) of the underlying optimization problem. Gradient-guided optimization (*e.g.*, gradient descent) is a promising alternative approach that has been shown to significantly outperform evolutionary algorithms at solving high-dimensional structured optimization problems in diverse domains including aerodynamic computations and machine learning [15], [16], [17].

However, gradient-guided optimization algorithms cannot be directly applied to fuzzing real-world programs as they often contain significant amounts of discontinuous behaviors (cases where the gradients cannot be computed accurately) due to widely different behaviors along different program branches [18], [19], [20], [21], [22]. We observe that this problem can be overcome by creating a smooth (*i.e.*, differentiable) surrogate function approximating the target program's branching behavior with respect to program inputs. Unfortunately, existing program smoothing techniques [19], [21] incur prohibitive performance overheads as they depend heavily on symbolic analysis that does not scale to large programs due to several fundamental limitations like path explosion, incomplete environment modeling, and large overheads of symbolic memory modeling [23], [24], [25], [26], [27], [28], [29].

In this paper, we introduce a novel, efficient, and scalable program smoothing technique using feed-forward Neural Networks (NNs) that can incrementally learn smooth approximations of complex, real-world program branching behaviors, *i.e.*, predicting the control flow edges of the target program exercised by a particular given input. We further propose a gradient-guided search strategy that computes and leverages the gradient of the smooth approximation (*i.e.*, an NN model) to identify target mutation locations that can maximize the number of detected bugs in the target program. We demonstrate how the NN model can be refined by incrementally retraining the model on mispredicted program behaviors. We find that feed-forward NNs are a natural fit for our task because of (i) their demonstrated ability to approximate complex non-linear functions, as implied by the universal approximation theorem [30], and (ii) their support for efficient and accurate computation of gradients/higher-order derivatives [17].

We design and implement our technique as part of NEUZZ, a new learning-enabled fuzzer. We compare NEUZZ with 10 state-of-the art fuzzers on 10 real-world programs covering 6 different file formats, (*e.g.*, ELF, PDF, XML, ZIP, TTF, and JPEG) with an average of 47,546 lines of code, the LAVA-M bug dataset [31], and the CGC dataset [32]. Our results show that NEUZZ consistently outperforms all the other fuzzers by a wide margin both in terms of detected bugs and achieved edge coverage. NEUZZ found 31 previously unknown bugs in the tested programs that other fuzzers failed to find. Our tests on the DARPA CGC dataset also confirmed that NEUZZ can outperform state-of-the-art fuzzers like Driller [10] at finding different bugs.

Our primary contributions in this paper are as follows:

- We are the first, to the best of our knowledge, to identify the significance of program smoothing for adopting efficient gradient-guided techniques for fuzzing.
- We introduce, to the best of our knowledge, the first efficient and scalable program smoothing technique using surrogate neural networks to effectively model the target program's branching behaviors. We further propose an incremental learning technique to iteratively refine the surrogate model as more training data becomes available.
- We demonstrate that the gradients of the surrogate neural network model can be used to efficiently generate program inputs that maximize the number of bugs found in the target program.
- We design, implement, and evaluate our techniques as part of NEUZZ and demonstrate that it significantly outperforms 10 state-of-the-art fuzzers on a wide range of real-world programs as well as curated bug datasets.

The rest of the paper is organized as follows. Section II summarizes the necessary background information on optimization and gradient-guided techniques. Section III provides an overview of our technique along with a motivating example. Section IV and Section V describe our methodology and implementation in detail. We present our experimental results in Section VI and describe some sample bugs found by NEUZZ in Section VII. Section VIII summarizes the related work and Section IX concludes the paper.

II. OPTIMIZATION BASICS

In this section, we first describe the basics of optimization and the benefits of gradient-guided optimization over evolutionary guidance for smooth functions. Finally, we demonstrate how fuzzing can be cast as an optimization problem.

An optimization problem usually consists of three different components: a vector of parameters x, an objective function $F(x)$ to be minimized or maximized, and a set of constraint functions $C_i(x)$ each involving either inequality or equality that must be satisfied. The goal of the optimization process is to find a concrete value of the parameter vector x that maximizes/minimizes $F(x)$ while satisfying all constraint functions $C_i(x)$ as shown below.

$$\underset{x \in R^n}{max/min}\ F(x) \text{ subject to } \begin{cases} C_i(x) \geq 0, i \in N \\ C_i(x) = 0, i \in Q \end{cases} \quad (1)$$

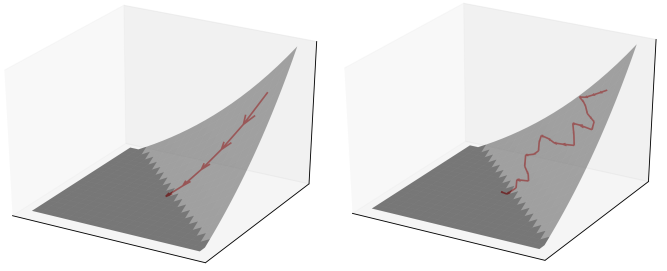
Here R, N, and Q denote the sets of real numbers, the indices for inequality constraints, and the indices for equality constraints, respectively.

Function smoothness & optimization. Optimization algorithms usually operate in a loop beginning with an initial guess of the parameter vector x and gradually iterating to find better solutions. The key component of any optimization algorithm is the strategy it uses to move from one value of x to the next. Most strategies leverage the values of the objective function F, the constraint functions C_i, and, if available, the gradient/higher-order derivatives.

The ability and efficiency of different optimization algorithms to converge to the optimal solution heavily depend on the nature of the objective and constraint functions F and C_i. In general, smoother functions (*i.e.*, those with well-defined and computable derivatives) can be more efficiently optimized than functions with many discontinuities (*e.g.*, ridges or plateaus). Intuitively, the smoother the objective/constraint functions are, the easier it is for the optimization algorithms to accurately compute gradients or higher-order derivatives and use them to systematically search the entire parameter space.

For the rest of this paper, we specifically focus on unconstrained optimization problems that do not have any constraint functions, *i.e.*, $C = \phi$, as they closely mimic fuzzing, our target domain. For unconstrained smooth optimization problems, gradient-guided approaches can significantly outperform evolutionary strategies at solving high-dimensional structured optimization problems [15], [16], [17]. This is because gradient-guided techniques effectively leverage gradients/higher-order derivatives to efficiently converge to the optimal solution as shown in Figure 1.

Convexity & gradient-guided optimization. For a common class of functions called convex functions, gradient-guided techniques are highly efficient and can always converge to the globally optimal solution [33]. Intuitively, a function is convex if a straight line connecting any two points on the graph of the function lies entirely above or on the graph. More formally, a function f is called convex if the following



(a) **gradient descent** (b) **evolutionary algorithm**

Fig. 1: **Gradient-guided optimization algorithms like gradient descent can be significantly more efficient than evolutionary algorithms that do not use any gradient information**

property is satisfied by all pairs of points x and y in its domain: $f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y), \forall t \in [0, 1]$.

However, in non-convex functions, gradient-guided approach may get stuck at locally optimal solutions where the objective function is greater (assuming that the goal is to maximize) than all nearby feasible points but there are other larger values present elsewhere in the entire range of feasible parameter values. However, even for such cases, simple heuristics like restarting the gradient-guided methods from new randomly chosen starting points have been shown to be highly effective in practice [17], [33].

Fuzzing as an unconstrained optimization. Fuzzing can be represented as an unconstrained optimization problem where the objective is to maximize the number of bugs/vulnerabilities found in the test program for a fixed number of test inputs. Therefore, the objective function can be thought of as $F_p(x)$, which returns 1 if input x triggers a bug/vulnerability when the target program p is executed with input x. However, such a function is too ill-behaved (*i.e.*, mostly containing flat plateaus and a few very sharp transitions) to be optimized efficiently.

Therefore, most graybox fuzzers instead try to maximize the amount of tested code (*e.g.*, maximize edge coverage) as a stand-in proxy metric [1], [8], [34], [35], [22]. Such an objective function can be represented as $F'_p(x)$ where F' returns the number of new control flow edges covered by the input x for program P. Note that F' is relatively easier to optimize than the original function F as the number of all possible program inputs exercising new control flow edges tend to be significantly higher than the inputs that trigger bugs/security vulnerabilities.

Most existing graybox fuzzers use evolutionary techniques [1], [8], [34], [35], [22] along with other domain-specific heuristics as their main optimization strategy. The key reason behind picking such algorithms over gradient-guided optimization is that most real-world programs contain many discontinuities due to significantly different behaviors along different program paths [36]. Such discontinuities may cause the gradient-guided optimization to get stuck at non-optimal solutions. In this paper, we propose a new technique using a neural network for smoothing the target programs to make them suitable for gradient-guided optimization and demonstrate how fuzzers might exploit such strategies to significantly boost their effectiveness.

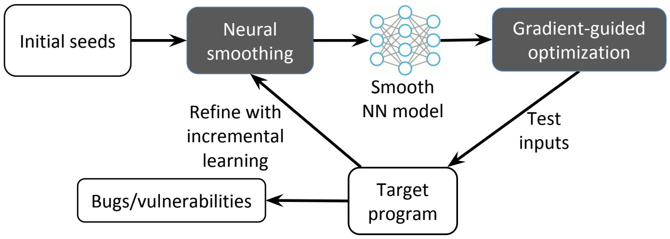


Fig. 2: **An overview of our approach**

III. Overview of Our Approach

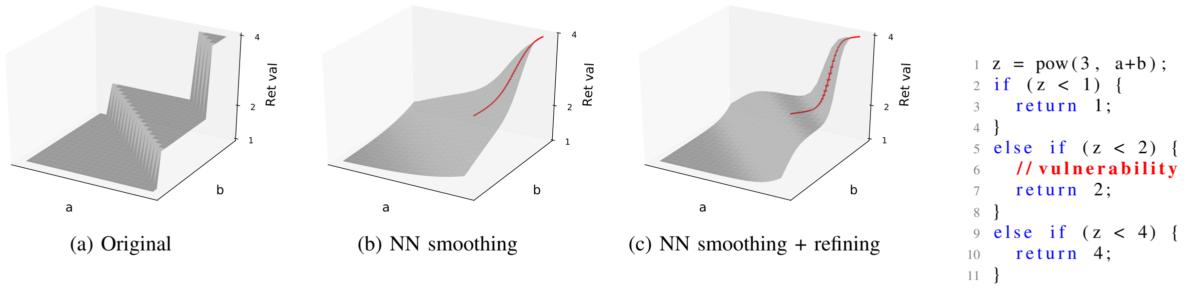
Figure 2 presents a high level overview of our approach. We describe the key components in detail below.

Neural program smoothing. Approximating a program's discontinuous branching behavior smoothly is essential for accurately computing gradients or higher-order derivatives that are necessary for gradient-guided optimization. Without such smoothing, the gradient-guided optimization process may get stuck at different discontinuities/plateaus. The goal of the smoothing process is to create a smooth function that can mimic a program's branching behavior without introducing large errors (*i.e.*, it deviates minimally from the original program behavior). We use a feed-forward neural network (NN) for this purpose. As implied by the universal approximation theorem [30], an NN is a great fit for approximating arbitrarily complex (potentially non-linear and non-convex) program behaviors. Moreover, NNs, by design, also support efficient gradient computation that is crucial for our purposes. We train the NN by either using existing test inputs or with the test input corpus generated by existing evolutionary fuzzers as shown in Figure 2.

Gradient-guided optimization. The smooth NN model, once trained, can be used to efficiently compute gradients and higher-order derivatives that can then be leveraged for faster convergence to the optimal solution. Different variants of gradient-guided algorithms like gradient descent, Newton's method, or quasi-Newton methods like the L-BFGS algorithm use gradients or higher-order derivatives for faster convergence [37], [38], [39]. Smooth NNs enable the fuzzing input generation process to potentially use all of these techniques. In this paper, we design, implement and evaluate a simple gradient-guided input generation scheme tailored for coverage-based fuzzing as described in detail in Section IV-C.

Incremental learning. Any types of existing test inputs (as long as they expose diverse behaviors in the target program) can be potentially used to train the NN model and bootstrap the fuzzing input generation process. In this paper, we train the NN by collecting a set of test inputs and the corresponding edge coverage information by running evolutionary fuzzers like AFL.

However, as the initial training data used for training the NN model may only cover a small part of the program space, we further refine the model through incremental training as new program behaviors are observed during fuzzing. The key



(a) Original

(b) NN smoothing

(c) NN smoothing + refining

```
1  z = pow(3, a+b);
2  if (z < 1) {
3     return 1;
4  }
5  else if (z < 2) {
6     //vulnerability
7     return 2;
8  }
9  else if (z < 4) {
10    return 4;
11 }
```

Fig. 3: **Simple code snippet demonstrating the benefits of neural smoothing for fuzzing**

challenge in incremental training is that if an NN is only trained on new data, it might completely forget the rules it learned from old data [40]. We avoid this problem by designing a new coverage-based filtration scheme that creates a condensed summary of both old and new data, allowing the NN to be trained efficiently on them.

A Motivating Example. We show a simple motivating example in Figure 3 to demonstrate the key insight behind our approach. The simple `C` code snippet shown in Figure 3 demonstrates a general switch-like code pattern commonly found in many real-world programs. In particular, the example code computes a non-linear exponential function of the input (*i.e.*, `pow(3,a+b)`). It returns different values based on the output range of the computed function. Let us also assume that a buggy code block (marked in **red**) is exercised if the function output range is in (1,2).

Consider the case where evolutionary fuzzers like AFL have managed to explore the branches in lines 2 and 9 but fail to explore branch in line 5. The key challenge here is to find values of `a` and `b` that will trigger the branch at line 5. Evolutionary fuzzers often struggle with such code as the odds of finding a solution through random mutation are very low. For example, Figure 3a shows the original function that the code snippet represents. There is a sharp jump in the function surface from $a + b = 0$ to $a + b - \epsilon = 0$ ($\epsilon \rightarrow +0$). To maximize the edge coverage during fuzzing, an evolutionary fuzzer can only resort to random mutations to the input as such techniques do not consider the shape of function surface. By contrast, our NN smoothing and gradient-guided mutations are designed to exploit the function surface shape as measured by the gradients.

We train an NN model on the program behaviors from the other two branches. The NN model smoothly approximates the program behaviors as shown in Figure 3b and 3c. We then use the NN model to perform more effective gradient-guided optimization to find the desired values of a and b and incrementally refine the model until the desired branch is found that exercises the target bug.

IV. Methodology

We describe the different components of our scheme in detail below.

A. *Program smoothing*

Program smoothing is an essential step to make gradient-guided optimization techniques suitable for fuzzing real-world programs with discrete behavior. Without smoothing, gradient-guided optimization techniques are not very effective for optimizing non-smooth functions as they tend to get stuck at different discontinuities [18]. The smoothing process minimizes such irregularities and therefore makes the gradient-guided optimization significantly more effective on discontinuous functions.

In general, the smoothing of a discontinuous function f can be thought of as a convolution operation between f and a smooth mask function g to produce a new smooth output function as shown below. Some examples of popular smoothing masks include different Gaussian and Sigmoid functions.

$$f'(x) = \int_{-\infty}^{+\infty} f(a)g(x-a)da \qquad (2)$$

However, for many practical problems, the discontinuous function f may not have a closed-form representation and thus analytically computing the above-mentioned integral is not possible. In such cases, the discrete version $f'(x) = \sum_a f(a)g(x-a)$ is used and the convolution is computed numerically. For example, in image smoothing, often fixed-sized 2-D convolution kernels are used to perform such computation. However, in our setting, f is a computer program and therefore the corresponding convolution cannot be computed analytically.

Program smoothing techniques can be classified into two broad categories: blackbox and whitebox smoothing. The blackbox approach picks discrete samples from the input space of f and computes the convolution numerically using these samples. By contrast, the whitebox approach looks into the program statements/instructions and try to summarize their effects using symbolic analysis and abstract interpretation [19], [21]. The blackbox approaches may introduce large approximation errors while whitebox approaches incur prohibitive performance overhead, which makes them infeasible for real-world programs.

To avoid such problems, we use NNs to learn a smooth approximation of program behaviors in a graybox manner (*e.g.*, by collecting edge coverage data) as described below.

B. Neural program smoothing

In this paper, we propose a novel approach to program smoothing by using surrogate NN models to learn and iteratively refine smooth approximations of the target program based on the observed program behaviors. The surrogate neural networks can smoothly generalize to the observed program behaviors while also accurately modeling potentially non-linear and non-convex behaviors. The neural networks, once trained, can be used for efficiently computing gradients and higher-level derivatives to guide the fuzzing input generation process as shown in Figure 3.

Why NNs? As implied by the universal approximation theorem [30], an NN is a great fit for approximating complex (potentially non-linear and non-convex) program behaviors. The advantages of using NNs for learning smooth program approximations are as follows: (i) NNs can accurately model complex non-linear program behaviors and can be trained efficiently. Prior works on model-based optimization have used simple linear and quadratic models [41], [42], [43], [44]. However, such models are not a good fit for modeling real-world software with highly non-linear and non-convex behaviors; (ii) NNs support efficient computation of their gradients and higher-order derivatives. Therefore, the gradient-guided algorithms can compute and use such information during fuzzing without any extra overhead; and (iii) NNs can generalize and learn to predict a program's behaviors for unseen inputs based on its behaviors on similar inputs. Therefore, NNs can potentially learn a smooth approximation of the entire program based on its behaviors for a small number of input samples.

NN Training. While NNs can be used to model different aspects of a program's behavior, in this paper we use them specifically for modeling the target program's branching behavior (*i.e.*, predicting control flow edges exercised by a given program input). One of the challenges in using neural nets to model branching behavior is the need to accept variably-sized input. Feedforward NNs, unlike real-world programs, typically accept fixed size input. Therefore, we set a maximum input size threshold and pad any smaller-sized inputs with null bytes during training. Note that supporting larger inputs is not a major concern as modern NNs can easily scale to millions of parameters. Therefore, for larger programs, we can simply increase the threshold size, if needed. However, we empirically find that relatively modest threshold values yield the best results and larger inputs do not increase modeling accuracy significantly.

Formally, let $f : \{\texttt{0x00}, \texttt{0x01}, ..., \texttt{0xff}\}^m \rightarrow \{0,1\}^n$ denote the NN that takes program inputs as byte sequences with size m and outputs an edge bitmap with size n. Let θ denote the trainable weight parameters of f. Given a set of training samples (X, Y), where X is a set of input bytes and Y represents the corresponding edge coverage bitmap, the training task of the parametric function $f(x, \theta) = y$ is to obtain the parameter $\hat{\theta}$ such that $\hat{\theta} = \arg\min_{\theta} \sum_{x \in X, y \in Y} L(y, f(x, \theta))$ where $L(y, f(x, \theta))$ defines the loss function between the output of the NN and the ground truth label $y \in Y$ in the training set. The training task is to find the weight parameters θ of the NN f to minimize the loss, which is defined using a distance metric. In particular, we use binary cross-entropy to compute the distance between the predicted bitmap and the true coverage bitmap. In particular, let y_i and $f_i(x, \theta)$ denote the i-th bit in the output bitmap of ground truth and f's prediction, respectively. Then, the binary cross-entropy between these two is defined as:

$$-\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot log(f_i(x,\theta) + (1-y_i) \cdot log(1-f_i(x,\theta)]$$

In this paper, we use feed-forward fully connected NNs to model the target program's branching behavior. The feed-forward architecture allows highly efficient computation of gradients and fast training [45].

Our smoothing technique is agnostic to the source of the training data and therefore the NN can be trained on any edge coverage data gathered from an existing input corpus. For our prototype implementation, we use input corpora generated by existing evolutionary fuzzers like AFL to train our initial model.

Training data preprocessing. Edge coverage exercised by the training data often tends to be biased, as it only contains labels for a small section of all edges in a program. For example, some edges might always be exercised together by all inputs in the training data. This type of correlation between a set of labels is known in machine learning as multicollinearity, which often prevents the model from converging to a small loss value [46]. To avoid such cases, we follow the common machine learning practice of dimensionality reduction by merging the edges that always appear together in the training data into one edge. Furthermore, we only consider the edges that have been activated at least once in the training data. These steps significantly reduce the number of labels to around $4,000$ from around $65,536$ on average. Note that we rerun the data preprocessing step at every iteration of incremental learning and thus some merged labels may get split as their correlation may decrease as new edge data is discovered during fuzzing.

C. Gradient-guided optimization

Different gradient-guided optimization techniques like gradient descent, Newton's method, or quasi-Newton methods like L-BFGS can use gradient or higher-order derivatives for faster convergence [37], [38], [39]. Smooth NNs enable the fuzzing input generation process to potentially use any of these techniques by supporting efficient computation of gradient and higher-order derivatives. In this paper, we specifically design a simple gradient-guided search scheme that is robust to minor prediction errors to demonstrate the effectiveness of our approach. We leave the exploration of more sophisticated techniques as future work.

Before describing our mutation strategy, which is based on the NN's gradient, we first provide a formal definition of the gradient, which indicates how much each input byte should be changed to affect the output of a final layer neuron in the NN (indicating changed edge coverage in the program) f [47]. Here each output neuron corresponds to a particular edge and computes the confidence value of whether a particular edge is covered given input bytes. The gradients of the output neurons of the NN f w.r.t. the inputs have been extensively used for adversarial input generation [48], [49] and visualizing/understanding DNNs [50], [47], [51]. Intuitively, in our setting, the goal of gradient-based guidance is to find inputs that will change the output of the final layer neurons corresponding to unseen edges from 0 to 1.

Given a parametric NN $y = f(\theta, x)$ as defined in Section IV-B, let y_i denote the output of i-th neuron in the final layer of f, which can also be written as $f_i(\theta, x)$. The gradient G of $f_i(\theta, x)$ with respect to input x can be defined as $G = \nabla_x f_i(\theta, x) = \partial y_i / \partial x$. Note that f's gradient w.r.t to θ can be easily computed as the NN training process requires iteratively computing this value to update θ. Therefore, G can also be easily calculated by simply replacing the computation of the gradient of θ to that of x. Note that the dimension of the gradient $\boldsymbol{G}$ is identical to that of the input x, which is a byte sequence in our case.

Algorithm 1 Gradient-guided mutation

```
Input:  seed ← initial seed
        iter ← number of iterations
        k ← parameter for dynamic top-k critical bytes
        g ← computed gradient of seed
 1: for $i = 1$ to $iter$ do
 2:     $locations \leftarrow top(g, k^i)$
 3:     for $m = 1$ to 255 do
 4:         for $loc \in locations$ do
 5:             $v \leftarrow seed[loc] + m * sign(g[loc])$
 6:             $v \leftarrow clip(v, 0, 255)$
 7:             $gen\_mutate(seed, loc, v)$
 8:         for $loc \in locations$ do
 9:             $v \leftarrow seed[loc] - m * sign(g[loc])$
10:             $v \leftarrow clip(v, 0, 255)$
11:             $gen\_mutate(seed, loc, v)$
```

Gradient-guided optimization. Algorithm 1 shows the outline of our gradient-guided input generation process. The key idea is to identify the input bytes with highest gradient values and mutate them, as they indicate higher importance to the NN and thus have higher chances of causing major changes in the program behavior (*e.g.*, flipping branches).

Starting from a seed, we iteratively generate new test inputs. As shown in Algorithm 1, at each iteration, we first leverage the absolute value of the gradient to identify the input bytes that will cause the maximum change in the output neurons corresponding to the untaken edges. Next, we check the sign of the gradient for each of these bytes to decide the direction of the mutation (*e.g.*, increment or decrement their values) to maximize/minimize the objective function. Conceptually, our usage of gradient sign is similar to the adversarial input generation methods introduced in [48]. We also bound the mutation of each byte in its legal range (*i.e.*, 0-255). Line 6 and line 10 denote the use of `clip` function to implement this.

We start the input generation process with a small mutation target (k in Algorithm 1) and exponentially grow the number of target bytes to mutate to effectively cover the large input space.

D. Refinement with incremental learning

The efficiency of the gradient-guided input generation process depends heavily on how accurately the surrogate NN can model the target program's branching behavior. To achieve higher accuracy, we incrementally refine the NN model when divergent program behaviors are observed during the fuzzing process (*i.e.*, when the target program's behavior does not match the predicted behavior). We use incremental learning techniques to keep the NN model updated by learning from new data when new edges are triggered.

The main challenge behind NN refinement is preventing the NN model from abruptly forgetting the information it previously learned from old data while training on new data. Such forgetting is a well-known phenomenon in deep learning literature and has been thought to be a result of the stability-plasticity dilemma [52], [53]. To avoid such forgetting issues, an NN must change the weights enough to learn new tasks but not too much as to cause it to forget previously learned representations.

The simplest way to refine an NN is to add the new training data (*i.e.*, program branching behaviors) together with the old data and train the model from scratch again. However, as the number of data points grows, such retraining becomes harder to scale. Prior research has tried to solve this problem using mainly two broad approaches [54], [55], [56], [57], [58], [59], [60]. The first one tries to keep separate representations for the new and old models to minimize forgetting using distributed models, regularization, or creating an ensemble out of multiple models. The second approach maintains a summary of the old data and retrains the model on new data along with the summarized old data and therefore is more efficient than complete retraining. We refer the interested readers to the survey by Kemker et al. [61] for more details.

In this paper, we used edge-coverage-based filtering to only keep the old data that triggered new branches for retraining. As new training data becomes available, we identify the ones achieving new edge coverage, put them together with the filtered old training data, and retrain the NN. Such a method effectively prevents the number of training data samples from drastically increasing over the number of retraining iterations. We find that our filtration scheme can easily support up to 50 iterations of retraining while still keeping the training time under several minutes.

V. Implementation

In this section, we discuss our implementation and how we fine-tune NEUZZ to achieve optimal performance. All our measurements are performed on a system running Arch Linux 4.9.48 with an Nvidia GTX 1080 Ti GPU.

TABLE I: NEUZZ Parameter Tuning

(a) **Edge coverage achieved by mutations generated in different iterations (Algorithm 1 line 1). The numbers in bold indicate the highest values for each program.**

Programs	Iteration i		
	7	10	11
readelf -a	1,678	**1,800**	1,529
libjpeg	**107**	89	93
libxml	161	**256**	174
mupdf	**294**	266	266

(b) **Edge coverage comparison of 1M mutations generated by NEUZZ on different NN models. n denotes the number of neurons in every hidden layer.**

Programs	1 hidden layer		3 hidden layers	
	n=4096	n=8192	n=4096	n=8192
readelf -a	1,800	1,658	1,714	1,584
libjpeg	89	57	80	79
libxml	256	172	140	99
mupdf	260	94	82	88

NN architecture. Our NN model is implemented in Keras-2.1.3 [62] with Tensorflow-1.4.1 [63] as a backend. The NN model consists of three fully-connected layers. The hidden layer uses ReLU as its activation function. We use sigmoid as the activation function for the output layer to predict whether a control flow edge is covered or not. The NN model is trained for 50 epochs (*i.e.*, 50 complete passes of the entire dataset) to achieve high test accuracy (around 95% on average). Since we use a simple feed-forward network, the training time for all 10 programs is less than 2 minutes. Even with pure CPU computation on an Intel i7-7700 running at 3.6GHz, the training time is under 20 minutes.

Training Data Collection. For each program tested, we run AFL-2.5.2 [1] on a single core machine for an hour to collect training data for the NN models. The average number of training inputs collected for 10 programs is around $2K$. The resulting corpus is further split into training and testing data with a 5:1 ratio, where the testing data is used to ensure that the models are not overfitting. We use `10KB` as the threshold file size for selecting our training data from the AFL input corpus (on average 90% of the files generated by AFL were under the threshold).

Mutation and Retraining. As shown in Figure 2, NEUZZ runs iteratively to generate 1M mutations and incrementally retrain the NN model. We first use the mutation algorithm described in Algorithm 1 to generate 1M mutations. We set the parameter `i` to 10, which generates 5,120 mutated inputs for a seed input. Next, we randomly choose 100 output neurons representing 100 unexplored edges in the target program and generate 10,240 mutated inputs from two seeds. Finally, we execute the target program with 1M mutated inputs using AFL's fork server technique [64] and use any inputs covering new edges for incremental retraining.

Model Parameter Selection. The success of NEUZZ depends on the choices of different parameters in training the models and generating mutations. Here, we empirically explore the optimal parameters that ensure maximum edge coverage on four programs: `readelf`, `libjpeg`, `libxml`, and `mupdf`. The results are summarized in Table I.

First, we evaluate how many critical bytes need to be mutated per initial seed (parameter k^i in Algorithm 1 line 1). We choose $k = 2$ as described in Section IV-C and show the coverage achieved by three iterations ($i = 7, 10, 11$ in Algorithm 1 line 1) with 1M mutations per iteration. For all four programs, *smaller* mutations (with fewer bytes changed per mutation) may lead to higher code coverage, as shown in Table Ia. The largest value of $i = 11$ achieves the least code coverage for all four programs. This result is potentially due to lines 4 and 8 in Algorithm 1—wasting too many mutations (out of the 1M mutation budget) on a single seed, without trying other seeds. However, the optimal number of mutation bytes varies across the four programs. For `readelf` and `libxml`, the optimal value of i is 10, while it is 7 for `libjpeg` and `mupdf`. Since the difference in achieved code coverage between $i = 7$ and $i = 10$ is not large, we choose $i = 10$ for the remainder of the experiments.

Next, we evaluate the choice of hyper-parameters in the NN model by varying the number of layers and the number of neurons in each hidden layer. In particular, we compare NN architectures with 1 and 3 hidden layers and 4096 and 8192 neurons per layer, respectively. For every target program, we use the same training data to train four different NN models and generate 1M mutations to test the achieved edge coverage. For all four programs, we find that the model with 1 hidden layer performs better than the one with 3 hidden layers. We think this is because the 1 hidden layer model is sufficiently complex to model the branching behavior of the target program, whereas the larger model (*i.e.*, with 3 hidden layers) is relatively harder to train and also tends to overfit.

VI. EVALUATION

In this section, we evaluate NEUZZ's bug finding performance and achieved edge coverage with respect to other state-of-the-art fuzzers. Specifically, we answer the following four research questions:

- **RQ1.** Can NEUZZ find more bugs than existing fuzzers?
- **RQ2.** Can NEUZZ achieve higher edge coverage than existing fuzzers?
- **RQ3.** Can NEUZZ perform better than existing RNN-based fuzzers?
- **RQ4.** How do different model choices affect NEUZZ's performance?

We start by describing our study subjects and experimental setting.

TABLE II: **Study Subjects**

(a) **Studied Fuzzers**

Fuzzer	**Technical Description**
AFL [1]	evolutionary search
AFLFast [8]	evolutionary + markov-model-based search
Driller [10]‡	evolutionary + concolic execution
VUzzer [34]	evolutionary + dynamic-taint-guided search
KleeFL [65]	evolutionary + seeds generated by symbolic execution
AFL-laf-intel [66]	evolutionary + transformed compare instruction
RNNfuzzer [67]	evolutionary + RNN-guided mutation filter
Steelix [35]†	evolutionary + instrumented comparison instruction
T-fuzz [68]†	evolutionary + program transformation
Angora [22]†	evolutionary + dynamic-taint-guided + coordinate descent + type inference

† We only compare based on the reported LAVA-M results as they are either not open-source or do not scale to our test programs.

‡ We only compare based on CGC as Driller only supports CGC binaries.

(b) **Studied Programs**

Programs		**# Lines**	**NEUZZ train (s)**	**AFL coverage**	
Class	Name			1 hour	6 hour
binutils-2.30 ELF Parser	readelf -a	21,647	108	4,490	5,041
	nm -C	53,457	63	3,779	4,243
	objdump -D	72,955	104	5,196	5,339
	size	52,991	52	2,578	3,291
	strip	56,330	55	5,789	6,169
TTF	harfbuzz-1.7.6	9,853	94	82,79	9,565
JPEG	libjpeg-9c	8,857	56	3,117	3,191
PDF	mupdf-1.12.0	123,562	62	4,624	4,871
XML	libxml2-2.9.7	73,920	95	6,691	7,153
Zip	zlib-1.2.11	1,893	65	1,479	1,665

A. *Study Subjects*

We evaluate NEUZZ on three different types of datasets: (i) 10 real-world programs, as shown in Table IIb, (ii) LAVA-M [31], and (iii) the DARPA CGC dataset [32]. To demonstrate the performance of NEUZZ, we compare the edge coverage and number of bugs detected by NEUZZ to 10 state-of-the-art fuzzers, as shown in Table IIa.

B. *Experimental Setup*

Our experimental setup includes the following two steps: First, we run AFL for an hour to generate the initial seed corpus. Then, we run each fuzzer for a fixed time budget with the same initial seed corpus and compare their achieved edge coverage and the number of bugs found. Specifically, the time budgets for 10 real world programs, LAVA-M datasets and CGC datasets are 24 hours, 5 hours and 6 hours respectively. For evolutionary fuzzers, the seed corpus is used to initialize the fuzzing process. For learning-based fuzzers (*i.e.*, NEUZZ and RNN-based fuzzers), the same seed corpus is used to generate the training dataset. As for KleeFL, a hybrid tool consisting of Klee and AFL, we run Klee for an extra hour to generate additional seeds, then add them into the original seed corpus for the following 24 hours fuzzing process. Note that we only report the additional code covered by the mutated inputs of each fuzzer without including the coverage information from the initial seed corpus.

In RQ3, we evaluate and compare the performance of NEUZZ with that of the RNN-based fuzzers. The RNN-based fuzzers could take up to 20× longer training time than NEUZZ. However, to focus on the efficacy of these two mutation algorithms, we evaluate the edge coverage for a fixed amount of mutations to exclude the effect of these disparate training time. We also perform a standalone evaluation comparing the training time costs for these two models. In RQ4, we also evaluate the edge coverage for a fixed number of mutations to exclude the effect of varying training time cost across different models.

C. *Results*

RQ1. Can* NEUZZ *find more bugs than existing fuzzers?

To answer this RQ, we evaluate NEUZZ *w.r.t.* other fuzzers in three settings: (i) Detecting real-world bugs. (ii) Detecting injected bugs in LAVA-M dataset [31]. (iii) Detecting CGC bugs. We describe the results in details.

(i) Detecting real-world bugs. We compare the total number of bugs and crashes found by NEUZZ and other fuzzers on 24-hour running time given the same seed corpus. There are five different types of bugs found by NEUZZ and other fuzzers: out-of-memory, memory leak, assertion crash, integer overflow, and heap overflow. To detect memory bugs that would not necessarily lead to a crash, we compile program binaries with AddressSanitizer [69]. We measure the unique memory bugs found by comparing the stack traces reported by AddressSanitizer. For crashes that do not cause AddressSanitizer to generate a bug report, we examine the execution trace. The integer overflow bugs are found by manually analyzing the inputs that trigger an infinite loop. We further verify integer overflow bugs using undefined behavior sanitizer [70]. The results are summarized in Table III.

NEUZZ finds all 5 types of bugs across 6 programs. AFL, AFLFast, and AFL-laf-intel find 3 types of bugs—they do not find any integer overflow bugs. The other fuzzers only uncover 2 types of bugs (*i.e.*, memory leak and assertion crash). AFL can a heap overflow bug on program `size`, while NEUZZ can find the same bug and another heap overflow bug on program `nm`. In total, NEUZZ finds 2× more bugs than the second best fuzzer. Moreover, the integer-overflow vulnerability in `strip`, *only found by* NEUZZ, has already been confirmed and is getting fixed by the developers.

(ii) Detecting injected bugs in LAVA-M dataset. The LAVA dataset is created to evaluate the efficacy of fuzzers by providing a set of real-world programs injected with a large number of

TABLE III: **Number of real-world bugs found by 6 fuzzers. We only list the programs where the fuzzers find a bug.**

Programs	AFL	AFLFast	VUzzer	KleeFL	AFL-laf-intel	NEUZZ
Detected Bugs per Project						
readelf	4	5	5	3	4	16
nm	8	7	0	0	6	9
objdump	6	6	0	3	7	8
size	4	4	0	3	2	6
strip	7	5	2	5	7	20
libjpeg	0	0	0	0	0	1
Detected Bugs per Type						
out-of-memory	✓	✓	✗	✓	✓	✓
memory leak	✓	✓	✓	✓	✓	✓
assertion crash	✗	✓	✗	✗	✓	✓
interger overflow	✗	✗	✗	✗	✗	✓
heap overflow	✓	✗	✗	✗	✗	✓
Total	29	27	7	14	26	60

bugs [31]. LAVA-M is a subset of the LAVA dataset, consisting of 4 GNU coreutil programs `base64`, `md5sum`, `uniq`, and `who` injected with 44, 57, 28, and 2136 bugs, respectively. All the bugs are guarded by four-byte magic number comparisons. The bugs get triggered only if the condition is satisfied. We compare NEUZZ's performance at finding these bugs to other state-of-the-art fuzzers, as shown in Table IV. Following conventional practice [22], [31], we use 5-hour time budget for the fuzzers' runtime.

Triggering a magic number condition in the LAVA dataset is a hard task for a coverage-guided fuzzer because the fuzzer has to generate the exact combination of 4 continuous bytes out of 256^4 possible cases. To solve this problem, we used a customized LLVM pass to instrument the magic byte checks like Steelix [35]. But unlike Steelix, we leverage the NN's gradient to guide the input generation process to find an input that satisfies the magic check. We run AFL for an hour to generate the training data and use it to train an NN whose gradients identify the possible critical bytes triggering the first byte-comparison of a magic-byte condition. Next, we perform a locally exhaustive search on each byte adjacent to the first critical byte to solve each of the remaining three byte-comparisons with 256 tries. Therefore, we need one NN gradient computation to find the byte locations that affect the magic checking and $4 \times 256 = 1024$ trials to trigger each bug. For program `md5sum`, following the latest suggestion of the LAVA-M's authors [71], we further reduce the seed into a single line, which significantly boosts the fuzzing performance.

As shown in Table IV, NEUZZ finds all the bugs in programs `base64`, `md5sum`, and `uniq`, and the highest number of bugs for program `who`. Note that LAVA-M authors left some bugs unlisted in all 4 programs, so the total number of bugs found by NEUZZ is actually higher than the number of listed bugs, as shown in the result.

NEUZZ has two key advantages over the other fuzzers. First, NEUZZ breaks the search space into multiple manageable steps: NEUZZ trains the underlying NN on AFL generated data, uses the computed gradient to reach the first critical byte, and performs a local search around the found critical region. Second, as opposed to VUzzer, which leverages magic numbers hard-coded in the target binary to construct program inputs, NEUZZ's gradient-based searching strategy do not rely on any hard-coded magic number. Thus, it can find all the bugs in program `md5sum`, which performs some computations on the input bytes before the magic number checking causing VUzzer to fail. In comparison to Angora, the current state-of-the-art fuzzer for LAVA-M dataset, NEUZZ finds 3 more bugs in `md5sum`. Unlike Angora, NEUZZ uses NN gradients to trigger the complex magic number conditions more efficiently.

TABLE IV: **Bugs found by different fuzzers on LAVA-M datasets.**

	base64	md5sum	uniq	who
#Bugs	44	57	28	2,136
FUZZER	7	2	7	0
SES	9	0	0	18
VUzzer	17	1	17	50
Steelix	43	28	24	194
Angora	48	57	29	1,541
AFL-laf-intel	42	49	24	17
T-fuzz	43	49	26	63
NEUZZ	48	60	29	1,582

(iii) Detecting CGC bugs. The DARPA CGC dataset [72] consists of vulnerable programs used in the DARPA Cyber Grand Challenge. These programs are implemented as network services performing various tasks and aim to mirror real-world applications with known vulnerabilities. Every bug in the program is guarded by a number of sanity checks on the input. The dataset comes with a set of inputs as proof of vulnerabilities.

We evaluate NEUZZ, Driller, and AFL on 50 randomly chosen CGC binaries. As running each test binary for each fuzzer takes 6 hours to run on CPU/GPU and our limited GPU resources do not allow us to execute multiple instances in parallel, we randomly picked 50 programs to keep the total experiment time within reasonable bounds. Similar to LAVA-M, here we also run AFL for an hour to generate the training data and use it to train the NN. We provide the same random seed to all three fuzzers and let them run for six hours. NEUZZ uses the same customized LLVM pass used for the LAVA-M dataset to instrument magic checkings in CGC binaries.

The results (Table V) show that NEUZZ uncovers 31 buggy binaries out of 50 binaries, while AFL and Driller find 21 and 25, respectively. The buggy binaries found by NEUZZ include all those found by Driller and AFL. NEUZZ further found bugs in 6 new binaries that both AFL and Driller fail to detect.

We analyze an example program `CROMU_00027` (shown in Listing 1). This is an ASCII content server that takes a query from a client and serves the corresponding ASCII code. A null-pointer dereferencing bug is triggered after a user tries to set command as VISUALIZE. AFL failed to detect this bug within 6-hour time budget due to its inefficiency at guessing the magic string. Although Driller tries to satisfy such complex

TABLE V: **Bugs found by 3 fuzzers on 50 CGC binaries**

Fuzzers	AFL	Driller	NEUZZ
Bugs	21	25	31

```
int cgc_ReceiveCommand(CommandStruct *command, int *
    more_command) {
  ...
  if (cgc_strncmp(&buffer[1], "VISUALIZE", cgc_strlen("
    VISUALIZE")) == 0) {
      command->command = VISUALIZE;
      //vulnerable code
...
```

Listing 1: **`cgc_ReceiveCommand` function in CROMU_00027**

magic string checking by concolic execution, in this case it fails to find an input that satisfies the check. By contrast, NEUZZ can easily use the NN gradient to locate the critical bytes in the program input that affects the magic comparison and find inputs that satisfy the magic check.

> **Result 1:** NEUZZ *found 31 previously unknown bugs in 6 different programs that other fuzzers could not find.* NEUZZ *also outperforms the state-of-the-art fuzzers at finding LAVA-M and CGC bugs.*

RQ2. Can NEUZZ *achieve higher edge coverage than existing fuzzers?*

To investigate this question, we compare the fuzzers on 24-hour fixed runtime budget. This evaluation shows not only the total number of new edges found by fuzzers but also the speed of new edge coverage versus time.

TABLE VI: **Comparing edge coverage of NEUZZ *w.r.t.* other fuzzers for 24 hours runs.**

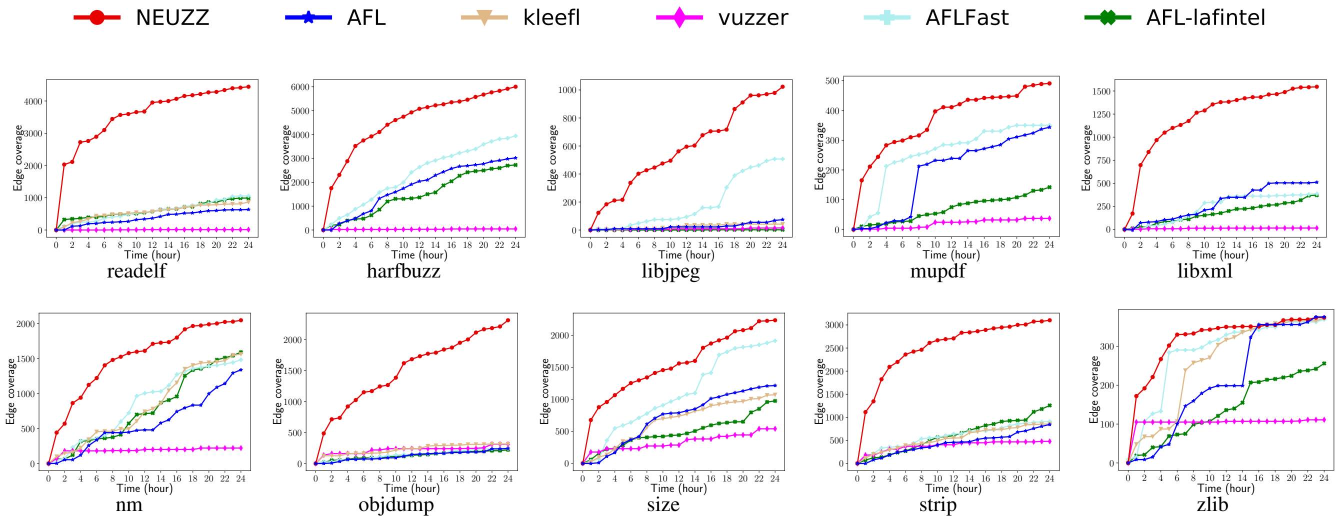
Programs	NEUZZ	AFL	AFLFast	VUzzer	KleeFL	AFL-laf-intel
readelf -a	4,942	746	1,073	12	968	1,023
nm -C	2,056	1,418	1,503	221	1,614	1,445
objdump -D	2,318	257	263	307	328	221
size	2,262	1,236	1,924	541	1,091	976
strip	3,177	856	960	478	869	1,257
libjpeg	1,022	94	651	60	67	2
libxml	1,596	517	392	16	n/a†	370
mupdf	487	370	371	38	n/a	142
zlib	376	374	371	15	362	256
harfbuzz	6,081	3,255	4,021	111	n/a	2,724

†indicates cases where Klee failed to run due to external dependencies

We collect the edge coverage information from AFL's edge coverage report. The results are summarized in Table VI. For all 10 real-world programs, NEUZZ significantly outperforms other fuzzers in terms of edge coverage. As shown in Fig 4, NEUZZ can achieve significantly more new edge coverage than other fuzzers within the first hour. On programs `strip`, `harfbuz` and `readelf`, NEUZZ can achieve more than $1,000$ new edge coverage *within an hour*. For programs `readelf` and `objdump`, the number of new edge coverage from NEUZZ's 1 hour running even beats the numbers of new edge coverage from all other fuzzers' 24 hours running. This shows the superior edge coverage ability of NEUZZ. For all 9 out of 10 programs, NEUZZ achieves 6×,1.5×,9×,1.8×,3.7×,1.9×,10×,1.3× and 3× edge coverage than baseline AFL, respectively, and 4.2×,1.3×,7×,1.2×,2.5×,1.5×,1.5×,1.3× and 3× edge coverage than the second highest number among all 6 fuzzers. For the smallest program `zlib`, which has less than 2k lines of code, NEUZZ achieves similar edge coverage with other fuzzers. We believe it reaches a saturation point when most of the possible edges for such a small program are already discovered after 24 hours fuzzing. The significant outperformance shows the effectiveness of NEUZZ in efficiently locating and mutating critical bytes using the gradient to cover new edges. NEUZZ also scales well in large systems. In fact, for programs with more than 10K lines (*e.g.*, `readelf`, `harfbuzz`, `mupdf` and `libxml`), NEUZZ achieves the highest edge coverage, where the taint-assisted fuzzer (*i.e.*, VUzzer) and symbolic execution assisted fuzzer (*i.e.*, KleeFL) either perform badly or does not scale.

The gradient-guided mutation strategy allows NEUZZ to explore diverse edges, while other evolutionary-based fuzzers often get stuck and repetitively check the same branch conditions. Also, the minimal execution overhead of the NN smoothing technique helps NEUZZ to scale well for larger programs while other advanced evolutionary fuzzers incur high execution overhead due to the use of heavyweight program analysis techniques like taint-tracking or symbolic execution.

Among the evolutionary fuzzers, AFLFast, uses an optimized seed selection strategies that focuses more on rare edges and thus achieves higher coverage than AFL on 8 programs, especially in `libjpeg`, `size` and `harfbuzz`. VUzzer, on the other hand, achieves higher coverage than AFL, AFLFast, and AFL-laf-intel within the first hour on small programs (*e.g.*, `zlib`, `nm`, `objdump`, `size` and `strip`), but its lead stalls quickly and eventually is surpassed by other fuzzers. Meanwhile, VUzzer's performance degrades on larger programs like `readelf`, `harfbuzz`, `libxml`, and `mupdf`. We suspect that the imprecision introduced by VUzzer's taint tracker causes it to perform poorly for large programs. KleeFL uses additional seeds generated by the symbolic execution engine Klee to guide AFL's exploration. Similar to VUzzer, for small programs (`nm`, `objdump`, and `strip`), KleeFL has good performance at the beginning, but its advantage of additional seeds from Klee would fade away after several hours. Moreover, KleeFL is based on Klee which cannot scale to large programs with complex library code, a well-known issue for symbolic execution. Thus, KleeFL does not have results on programs `libxml`, `mupdf` and `harfbuzz`. Unlike VUzzer and KleeFL, NEUZZ does not rely on any heavy program analysis techniques; NEUZZ uses the gradients computed from NNs to generate promising mutations even for larger programs. The efficient NN gradient computation process allow NEUZZ to scale better than VUzzer



readelf harfbuzz libjpeg mupdf libxml

nm objdump size strip zlib

Fig. 4: **The edge coverage of different fuzzers running for 24 hours.**

and KleeFL at identifying the critical bytes that affect different unseen program branches, achieving significantly more edge coverage.

AFL-laf-intel transforms complex magic number comparison into nested byte-comparison using an LLVM pass and then runs AFL on the transformed binaries. It achieves second-highest new edge coverage on program `strip`. However, the comparison transformations add additional instructions to common comparison operations and thus cause a potential edge explosion issue. The edge explosion would greatly increase the rate of edge conflict and hurt the performance of evolutionary fuzzing. Also, these additional instructions would cause extra execution overheads. As a result, programs like `libjpeg` with frequent comparison operations suffer significant slowdown (*e.g.*, `libjpeg`), and AFL-laf-intel struggles to trigger new edges.

> **Result 2:** NEUZZ *can achieve significantly higher edge coverage compared to other gray-box fuzzers (up to 4 × better than AFL, and 2.5× better than the second-best one for 24-hour running).*

RQ3. Can **NEUZZ** *perform better than existing RNN-based fuzzers?*

Existing recurrent neural network (RNN)-based fuzzers learn mutation patterns from past fuzzing experience to guide future mutations [67]. These models first learn mutation patterns (composed of critical bytes) from a large number of mutated inputs generated by AFL. Next, they use the mutation patterns to build a filter to AFL which only allows mutations on critical bytes to pass, vetoing all other non-critical byte mutations. We choose 4 programs studied by the previous work to evaluate the performance of NEUZZ compared to the RNN-based fuzzer for 1 million mutations. We train two NN models with the same training data, then let the two NN-based fuzzers run to generate 1 million mutations and compare the new code coverage achieved by the two methods. We report both the achieved edge coverage and training time, as shown in Table VII.

TABLE VII: **NEUZZ vs. RNN fuzzer *w.r.t.* baseline AFL**

Programs	Edge Coverage			Training Time (sec)		
	NEUZZ	RNN	AFL	NEUZZ	RNN	AFL
readelf -a	1,800	215	213	108	2,224	NA
libjpeg	89	21	28	56	1,028	NA
libxml	256	38	19	95	2,642	NA
mupdf	260	70	32	62	848	NA

For all the four programs, NEUZZ significantly outperforms the RNN-based fuzzer on 1M mutations. NEUZZ achieves $8.4\times, 4.2\times, 6.7\times$, and $3.7\times$ more edge-coverage than the RNN-based fuzzer across the four programs respectively. In addition, the RNN-based fuzzer has, on average, 20× more training overhead than NEUZZ, because RNN models are significantly more complicated than feed-forward network models.

An additional comparison of the RNN-based fuzzer with AFL shows that the former achieves 2× more edge coverage on average than AFL on `libxml` and `mupdf` using the 1-hour corpus. We also observe that the RNN-based fuzzer vetoes around 50% of the mutations generated by AFL. Thus, the new edge coverage of 1M mutations from RNN-based fuzzer can achieve the edge coverage of 2M mutations in vanilla AFL. This explains why the RNN-based fuzzer uncovers around 2× more new edges of AFL on some programs. If AFL gets stuck on 2M mutations, the RNN-based fuzzer would also get stuck on 1M filtered mutations.

The key advantage of NEUZZ over the RNN-based fuzzer is that NEUZZ obtains critical locations using neural-network-based gradient-guided search, while the RNN fuzzer tries to model the task in an end-to-end manner. Our model can distinguish different contributing factors of critical bytes that the RNN model may miss as demonstrated by our experimental results. For mutation generation, we perform an exhaustive search for critical bytes determined by corresponding contributing factors, while the RNN-based fuzzer still relies on AFL's uniform random mutations.

> **Result 3:** NEUZZ*, a fuzzer based on simple feed-forward network, significantly outperforms the RNN-based fuzzers by achieving 3.7× to 8.4× more edge coverage across different projects.*

RQ4. How do different model choices affect NEUZZ*'s performance?*

NEUZZ's fuzzing performance heavily depends on the accuracy of the trained NN. As described in Section V, we empirically find that an NN model with 1 hidden layer is expressive enough to model complex branching behavior of real-world programs. In this section, we conduct an ablation study by exploring different model settings for a 1 hidden layer architecture, *i.e.*, a linear model, an NN model without refinement, and an NN model with incremental refinement. We evaluate the effect of these models on NEUZZ's performance.

To compare the fuzzing performance, we generate 1M mutations for each version of NEUZZ on 4 programs. We implement the linear model by removing the non-linear activation functions used in the hidden layer and thus making the whole feed-forward network completely linear. The NN model is trained same seed corpus from AFL. Next, We generate 1M mutations from the passive learning model and measure the edge coverage achieved by these 1M mutations. Finally, we filter out the mutated inputs that exercise unseen edges from the 1 million mutations and add these selected inputs to original seed corpus to incrementally retrain another NN model and use it to generate further mutations. The results are summarized in Table VIII. We can see that both NN models (with or without incremental learning) outperform the linear models for all 4 tested programs. This shows that the nonlinear NN models can approximate program behaviors better than a simple linear model. We also observe that incremental learning helps NNs to achieve significantly higher accuracy and therefore higher edge coverage.

> **Result 4:** *NN models outperform linear models and incremental learning makes NNs even more accurate over time.*

VII. CASE STUDIES OF BUGS

In this section, we provide samples of and analyze three different types of bugs discovered by NEUZZ: integer overflow, out-of-memory bug, and crash-inducing bug.

We note that a large number of program bugs result from incorrect handling of extreme values of variables. As NEUZZ can enumerate all critical bytes from `0x00` to `0xff` (see Algorithm 1 line 3), we manage to find a large number of bugs caused by mishandled extrema. For example, NEUZZ is able to find many out-of-memory bugs in `libjpeg`, `objdump`, `nm` and `strip` by setting the input bytes that affect memory allocation size to extremely large values.

TABLE VIII: **Edge coverage comparison of 1M mutations generated by NEUZZ using different machine learning models.**

Programs	Linear Model	NN Model	NN + Incremental
readelf -a	1,723	1,800	2,020
libjpeg	63	89	159
libxml	117	256	297
mupdf	93	260	329

`strip`'s integer overflow. NEUZZ found an integer overflow bug that can induce an infinite loop on `strip`. Listing 2 shows a function in the `strip` program that parses every section in the program header table of an input ELF file and assigns all sections to a new program header table in the output ELF file. The integer overflow occurs at the if-condition in line 11 of Listing 2 as NEUZZ sets `segment_size` to an extremely large value. Consequently, the program gets stuck in an infinite loop. We found this bug exists in both the latest version of Binutils 2.30 and in older versions 2.26 and 2.29.

`libjpeg`'s out-of-memory. During the JPEG compression process, the data of every color space is down-sampled by the corresponding sampling factor in order to reduce file size. According to the JPEG standard, the sampling factor must be an integer between 1 and 4. This value is used during the decompression process to determine how much memory needs to be allocated as shown in Listing 4. NEUZZ sets a large value which causes too much memory to be allocated for image data, causing a out-of-memory error. Such errors can potentially be exploited to launch denial of service attacks on servers using `libjpeg` for displaying images.

```
1  // binutils-2.30/bfd/elf.c:6499
2  #define IS_CONTAINED(saddr, ssize, baddr) \
3    (saddr >= baddr
4     && saddr <= (baddr + ssize))
5
6  rewrite_elf_program_header (bfd *ibfd, bfd *obfd)
7  {
8    for(j = 0; j < section_count; j++)
9    {
10        output_section = section->output_section;
11        if(IS_CONTAINED(output_section, segment_size,
      base_addr))
12        {
13          ...
14          isec++;
15          sections[j] = NULL;
16          ...
17        }
18    }
19 }
```

Listing 2: `strip` integer overflow

```
// binutils-2.30/binutils/readelf.c:5901
static bfd_boolean
process_section_headers (Filedata * filedata)
{
  filedata->section_headers = NULL;
  ...
  if (filedata->file_header.e_shnum == 0)
  {
    ...
    return TRUE;
  }
}
// binutils-2.30/binutils/readelf.c:654
static Elf_Internal_Shdr *
find_section (Filedata * filedata, const char * name)
{
  ...
  assert (filedata->section_headers != NULL);
  ...
}
```

Listing 3: `readelf` section header parsing bug

```
// libjpeg/jmemmgr.c:444
(JBLOCKARRAY) alloc_barray (j_common_ptr cinfo, ...)
{
  ...
  while (currow < numrows) {
    ...
    alloc_large((size_t) rowsperchunk * (size_t)
   blocksperrow * SIZEOF(JBLOCK));
    ...
  }
}
```

Listing 4: `libjpeg` out-of-memory bug

`readelf`'s crash. An ELF file consists of a file header, program header, section header and section data. According to the ELF specification, the ELF header contains the field `e_shnum` located at the 60th byte for a 64-bit binary, which specifies the number of sections in the ELF file. NEUZZ sets the number of sections of the input file to be 0. As shown in Listing 3, if the number of sections is equal to 0, the implementation returns a `NULL` pointer which is dereferenced by subsequent code, triggering a crash.

VIII. Related Work

Program smoothing. Parnas et al. [18] observed that discontinuities are one of the fundamental challenges behind the development of secure and reliable software. Chaudhury et al. [19], [73], [36] suggested the idea of program smoothing to facilitate program analysis and presented a rigorous smoothing algorithm using abstract interpretation and symbolic execution. Unfortunately, such algorithms incur prohibitive performance overhead, especially for large programs. By contrast, the proposed smoothing technique leverages the learning power of NNs to achieve better scalability.

Learning-based fuzzing. Recently, there has been increasing interest in using machine learning techniques for improving fuzzers [74], [67], [75], [76], [77], [78], [79]. However, existing learning-based fuzzers model fuzzing as an end-to-end ML problem, *i.e.*, they learn ML models to directly predict input patterns that can achieve higher code coverage. By contrast, we first use NNs to smoothly approximate the program branching behavior and then leverage gradient-guided input generation technique to achieve higher coverage. Therefore, our approach is more tolerant to learning errors by ML models than the end-to-end approaches. In this paper, we empirically demonstrate that our strategy outperforms end-to-end modeling both in terms of finding bugs and achieving higher edge coverage [67].

Taint-based fuzzing. Several evolutionary fuzzers have tried to use taint information to identify promising mutating locations [80], [81], [82], [34], [35], [22]. TaintScope [80] is designed to identify input bytes that affects system/library calls and focus on mutating these bytes. Dowser [81] and BORG [82] target buffer boundary violations and buffer overread vulnerabilities respectively. Vuzzer [34] captures magic constants by static analysis and mutates existing values to these constants. Steelix [35] instruments binaries to collect additional taint information about comparing instructions. Angora [22] uses dynamic taint tracking to identify promising mutation locations and perform coordinate descent to guide mutations on these locations. However, all these taint-tracking-based approaches are fundamentally limited by the fact that dynamic taint analysis incurs very high overhead while static taint analysis suffers from a high rate of false positives. Our experimental results demonstrate that NEUZZ easily outperforms existing state-of-the-art taint-based fuzzers.

Several fuzzers and test input generators [20], [83], [22] have tried to use different forms of gradient-guided optimization algorithms directly on the target programs. However, without program smoothing, such techniques tend to struggle and get stuck at the discontinuities.

Symbolic/concolic execution. Symbolic and concolic execution [23], [25], [24], [84], [11] use Satisfiability Modulo Theory (SMT) solvers to solve path constraints and find interesting test inputs. Several projects have also tried to combining fuzzing with such approaches [85], [65], [10]. Unfortunately, these approaches struggle to scale in practice due to several fundamental limitations of symbolic analysis including path explosion, incomplete environment modeling, large overheads of symbolic memory modeling, etc. [26].

Neural programs. A neural program is essentially a neural network that learns a latent representation of the target program's logic. Several recent works have synthesized such neural programs from input-output samples of a program to accurately predict the program's outputs for new inputs [86], [87], [88]. By contrast, we use NNs to learn smooth approximations of a program's branching behaviors. Concurrent to our work, another work recently posted on arXiv has proposed using NN models to make symbolic execution scale better [89]. However, their goal and underlying techniques are very different from ours.

IX. Conclusion

We present NEUZZ, an efficient learning-enabled fuzzer that uses a surrogate neural network to smoothly approximate a target program's branch behavior. We further demonstrate how gradient-guided techniques can be used to generate new test inputs that can uncover different bugs in the target program. Our extensive evaluations show that NEUZZ significantly outperforms other 10 state-of-the-art fuzzers both in the numbers of detected bugs and achieved edge coverage. Our results demonstrate the vast potential of leveraging different gradient-guided input generation techniques together with neural smoothing to significantly improve the effectiveness of the fuzzing process.

References

[1] M. Zalewski, "American Fuzzy Lop (AFL) README," http://lcamtuf.coredump.cx/afl/README.txt, 2018.

[2] DARPA, "Cyber Grand Challenge," http://archive.darpa.mil/cybergrandchallenge/, 2016.

[3] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] "Google Fuzzing Service for OS Finds 1K Bugs in Five Months," http://www.eweek.com/cloud/google-fuzzing-service-for-os-finds-1k-bugs-in-five-months, 2017.

[5] "Guided in-process fuzzing of chrome components," https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html, 2016.

[6] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "KameleonFuzz: evolutionary fuzzing for black-box XSS detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014, pp. 37–48.

[7] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 2155–2168.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[9] K. Serebryany, "libFuzzer – a library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, 2018.

[10] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[11] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[12] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," *Proceedings of the IEEE Symposium on Security & privacy*, 2018.

[13] B. P. Miller, "Fuzz testing of application reliability," *UW-Madison Computer Sciences*, 2007.

[14] S. Hocevar, "zzuf—multi-purpose fuzzer," http://caca.zoy.org/wiki/zzuf, 2011.

[15] D. W. Zingg, M. Nemec, and T. H. Pulliam, "A comparative evaluation of genetic and gradient-based algorithms applied to aerodynamic optimization," *European Journal of Computational Mechanics/Revue Européenne de Mécanique Numérique*, vol. 17, no. 1-2, pp. 103–126, 2008.

[16] R. Horst and P. M. Pardalos, *Handbook of global optimization*. Springer Science & Business Media, 2013, vol. 2.

[17] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[18] D. Parnas, "Software aspects of strategic defense systems," *Communications of the ACM*, vol. 28, no. 12, pp. 1326–1335, 1985.

[19] S. Chaudhuri and A. Solar-Lezama, "Smoothing a program soundly and robustly," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 277–292.

[20] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[21] S. Chaudhuri and A. Solar-Lezama, "Smooth interpretation," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 279–291, 2010.

[22] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *arXiv preprint arXiv:1803.01307*, 2018.

[23] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[24] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30. ACM, 2005, pp. 263–272.

[25] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[26] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[27] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1066–1071.

[28] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40. ACM, 2005, pp. 213–223.

[29] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 553–568.

[30] K.-I. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural networks*, vol. 2, no. 3, pp. 183–192, 1989.

[31] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[32] DARPA, "Cyber Grand Challenge Repository," https://github.com/cybergrandchallenge, 2017.

[33] S. Wright and J. Nocedal, "Numerical optimization," *Springer Science*, vol. 35, no. 67-68, p. 7, 1999.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-Aware Evolutionary Fuzzing," in *Proceedings of the 2008 Network and Distributed Systems Security Conference*, 2017.

[35] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[36] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity and robustness of programs," *Communications of the ACM*, vol. 55, no. 8, pp. 107–115, 2012.

[37] D. P. Bertsekas and A. Scientific, *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[38] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A limited memory algorithm for bound constrained optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995.

[39] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

[40] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165.

[41] A. R. Conn, K. Scheinberg, and P. L. Toint, "Recent progress in unconstrained nonlinear optimization without derivatives," *Mathematical programming*, vol. 79, no. 1-3, p. 397, 1997.

[42] A. Conn, K. Scheinberg, and P. Toint, "On the convergence of derivative-free methods for unconstrained optimization," *Approximation theory and optimization: tributes to MJD Powell*, pp. 83–108, 1997.

[43] M. J. Powell, "Uobyqa: unconstrained optimization by quadratic approximation," *Mathematical Programming*, vol. 92, no. 3, pp. 555–582, 2002.

[44] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM review*, vol. 45, no. 3, pp. 385–482, 2003.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[46] A. Garg and K. Tai, "Comparison of statistical and machine learning methods in modelling of data with multicollinearity," *International Journal of Modelling, Identification and Control*, vol. 18, no. 4, pp. 295–312, 2013.

[47] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps." in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[48] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[49] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings." in *IEEE European Symposium on Security & Privacy*, 2015.

[50] J. Yosinski, J. Clune, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *2015 ICML Workshop on Deep Learning*, 2015.

[51] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[52] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165.

[53] W. C. Abraham and A. Robins, "Memory retention–the synaptic stability versus plasticity dilemma," *Trends in neurosciences*, vol. 28, no. 2, pp. 73–78, 2005.

[54] G. E. Hinton and D. C. Plaut, "Using fast weights to deblur old memories," in *Proceedings of the ninth annual conference of the Cognitive Science Society*, 1987, pp. 177–186.

[55] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, 2017.

[56] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra, "Pathnet: Evolution channels gradient descent in super neural networks," *arXiv preprint arXiv:1701.08734*, 2017.

[57] B. Ren, H. Wang, J. Li, and H. Gao, "Life-long learning based on dynamic combination model," *Applied Soft Computing*, vol. 56, pp. 398–404, 2017.

[58] T. J. Draelos, N. E. Miner, C. C. Lamb, J. A. Cox, C. M. Vineyard, K. D. Carlson, W. M. Severa, C. D. James, and J. B. Aimone, "Neurogenesis deep learning: Extending deep networks to accommodate new classes," in *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE, 2017, pp. 526–533.

[59] B. Goodrich and I. Arel, "Unsupervised neuron selection for mitigating catastrophic forgetting in neural networks," in *Proc. of the Int. Symp. on Circuits and Systems*. Citeseer, 2014, pp. 997–1000.

[60] A. Robins, "Catastrophic forgetting, rehearsal and pseudorehearsal," *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995.

[61] R. Kemker, A. Abitino, M. McClure, and C. Kanan, "Measuring catastrophic forgetting in neural networks," *arXiv preprint arXiv:1708.02072*, 2017.

[62] "Keras: The python deep learning library," https://keras.io/, 2018.

[63] "An open source machine learning framework for everyone," https://www.tensorflow.org/, 2018.

[64] lcamtuf, "Fuzzing random programs without execve()," https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014.

[65] J. Fietkau, B. Shastry, and J.-P. Seifert, "KleeFL - Seeding Fuzzers With Symbolic Execution," https://github.com/julieeen/kleefl, 2017.

[66] L. Intel, "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[67] M. Rajpal, W. Blum, and R. Singh, "Not All Bytes Are Equal: Neural Byte Sieve for Fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[68] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[69] "Addresssanitizer, threadsanitizer, memorysanitizer," https://github.com/google/sanitizers, 2018.

[70] "Undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2018.

[71] B. Dolan-Gavitt, "Of Bugs and Baselines," http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html, 2018.

[72] "Darpa challenges sets for linux, windows, and macos," https://github.com/trailofbits/cb-multios, 2017.

[73] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity analysis of programs," in *ACM Sigplan Notices*, vol. 45. ACM, 2010, pp. 57–70.

[74] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine Learning for Input Fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[75] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-Driven Seed Generation for Fuzzing," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017.

[76] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[77] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 521–538.

[78] K. Böttinger, P. Godefroid, and R. Singh, "Deep Reinforcement Fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[79] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[80] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[81] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[82] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The BORG: Nanoprobing Binaries for Buffer Overreads," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[83] L. Szekeres, "Memory corruption mitigation via hardening and testing," Ph.D. dissertation, Stony Brook University, 2017.

[84] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[85] S. K. Cha, M. Woo, and D. Brumley, "Program-Adaptive Mutational Fuzzing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[86] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[87] S. Reed and N. d. Freitas, "Neural Programmer-Interpreters," *arXiv preprint arXiv:1711.04596*, 2015.

[88] A. Neelakantan, Q. V. Le, M. Abadi, A. McCallum, and D. Amodei, "Learning a natural language interface with neural programmer," *arXiv preprint arXiv:1611.08945*, 2016.

[89] S. Shen, S. Ramesh, S. Shinde, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: The feasibility of an inductive approach to symbolic execution," *arXiv preprint arXiv:1807.00575*, 2018.